Relazione Tecnica - Progetto Server HTTP Concorrente

Laboratorio di Reti e Sistemi Distribuiti

Federico Celi
Matricola: 554255

Indice

1 Introduzione

La presente relazione descrive la progettazione e l'implementazione di un server HTTP concorrente sviluppato in linguaggio C. Questo server è in grado di accettare e gestire connessioni multiple da client tramite l'utilizzo dei thread POSIX (pthreads). L'applicazione è stata realizzata come parte del corso di Laboratorio di Reti e Sistemi Distribuiti dell'Università di Messina e rappresenta un esercizio completo di progettazione di un'applicazione di rete con finalità didattiche ma anche con caratteristiche concrete.

2 Obiettivi del Progetto

Gli obiettivi principali del progetto sono stati:

- Implementare un server TCP in linguaggio C utilizzando i socket BSD.
- Integrare la gestione concorrente delle connessioni tramite l'utilizzo di thread.
- Gestire correttamente i metodi HTTP di base: GET, POST, PUT, DELETE.
- Restituire risposte coerenti secondo lo standard HTTP/1.1.
- Familiarizzare con la gestione delle richieste da parte di client HTTP.

3 Fondamenti Teorici

3.1 Socket e Comunicazione TCP

Un socket è un endpoint di una comunicazione bidirezionale tra due processi, spesso tra client e server. I socket TCP permettono comunicazioni affidabili grazie al controllo dell'ordine e della consegna dei pacchetti. In ambiente Unix, i socket sono gestiti attraverso una serie di chiamate di sistema come `socket()`, `bind()`, `listen()`, `accept()`, `read()` e `write()` o `send()` e `recv()`.

3.2 Il Protocollo HTTP

HTTP (HyperText Transfer Protocol) è un protocollo basato su testo, utilizzato per la comunicazione tra client (come browser) e server web. La sua struttura si basa su richieste contenenti un metodo, un URI e una versione del protocollo. Ogni richiesta è seguita da zero o più intestazioni e, facoltativamente, da un corpo. I metodi più comuni includono:

- `GET`: recupera una risorsa.
- `POST`: invia dati per la creazione di una risorsa.
- `PUT`: invia dati per aggiornare una risorsa.
- `DELETE`: elimina una risorsa.

4 Analisi del Codice

Il file sorgente `server_http.c` contiene tutto il codice necessario per creare un server multithreaded che risponde a richieste HTTP.

4.1 Inclusione delle Librerie

Le librerie standard utilizzate permettono la gestione delle stringhe, della memoria, dei socket e dei thread.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>
```

4.2 Definizione di Costanti e Prototipi

```
#define PORT 8080
#define BUFFER_SIZE 4096

void *gestisci_connessione(void *arg);
void invia_risposta(int client_socket, const char *metodo);
```

4.3 Funzione main()

`main()` rappresenta il punto di ingresso del programma.

```
int main() {
    int server_fd, client_socket;
    struct sockaddr_in address;
    socklen_t addr_len = sizeof(address);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 0) {
        perror("Errore nella creazione del socket");
        exit(EXIT_FAILURE);
    }
```

4.4 Bind e Listen

```
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
        0) {
        perror("Errore nel bind");
        exit(EXIT_FAILURE);
```

```
    }

    if (listen(server_fd, 10) < 0) {
        perror("Errore nella listen");
        exit(EXIT_FAILURE);
    }
```

4.5 Accept e Threading

```
    while (1) {
        client_socket = accept(server_fd, (struct sockaddr *)&address, &
            addr_len);
        if (client_socket < 0) {
            perror("Errore nella accept");
            continue;
        }
        pthread_t tid;
        pthread_create(&tid, NULL, gestisci_connessione, (void *)&
            client_socket);
        pthread_detach(tid);
    }
    close(server_fd);
    return 0;
}
```

5 Funzione gestisci_connessione

```
void *gestisci_connessione(void *arg) {
    int client_socket = *(int *)arg;
    char buffer[BUFFER_SIZE] = {0};
    read(client_socket, buffer, BUFFER_SIZE);

    printf("Richiesta ricevuta:\n%s\n", buffer);
    char metodo[8];
    sscanf(buffer, "%s", metodo);
    invia_risposta(client_socket, metodo);
    close(client_socket);
    pthread_exit(NULL);
}
```

6 Funzione invia_risposta

```
void invia_risposta(int client_socket, const char *metodo) {
    char risposta[BUFFER_SIZE];

    if (strcmp(metodo, "GET") == 0) {
        sprintf(risposta, "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r
            \n\r\nRisposta GET ricevuta.");
    } else if (strcmp(metodo, "POST") == 0) {
        sprintf(risposta, "HTTP/1.1 201 Created\r\nContent-Type: text/
            plain\r\n\r\nRisorsa creata con POST.");
```

```
    } else if (strcmp(metodo, "PUT") == 0) {
        sprintf(risposta, "HTTP/1.1␣201␣No␣Content\r\n\r\nRisorsa␣
            aggiornata␣con␣PUT.");
    } else if (strcmp(metodo, "DELETE") == 0) {
        sprintf(risposta, "HTTP/1.1␣200␣OK\r\nContent-Type:␣text/plain\r
            \n\r\nRisorsa␣eliminata.");
    } else {
        sprintf(risposta, "HTTP/1.1␣400␣Bad␣Request\r\nContent-Type:␣
            text/plain\r\n\r\nMetodo␣non␣supportato.");
    }
    send(client_socket, risposta, strlen(risposta), 0);
}
```

7 Test Funzionali

Il funzionamento del server è stato testato utilizzando il comando `curl`, simulando richieste HTTP diverse.

```
curl -X GET http://localhost:8080
curl -X POST http://localhost:8080
curl -X PUT http://localhost:8080
curl -X DELETE http://localhost:8080
```

8 Possibili Estensioni e Migliorie

Il server, nella sua versione attuale, offre una base solida ma minimale per la gestione di richieste HTTP. Numerose funzionalità possono essere introdotte per migliorare le sue capacità, robustezza e usabilità. Di seguito vengono elencate alcune possibili estensioni che possono rappresentare sviluppi futuri:

- **Gestione delle Intestazioni HTTP**
- **Parsing del Corpo della Richiesta**
- **Logging su File**
- **Supporto per File Statici**
- **Interfaccia Web di Monitoraggio**
- **Autenticazione Base**
- **Sicurezza e Validazione**
- **Gestione delle Risorse Condivise**
- **Integrazione HTTPS con OpenSSL**

9 Conclusioni

Il progetto svolto ha rappresentato un'esperienza altamente formativa nel campo della programmazione di rete e della progettazione di software concorrente. Attraverso lo sviluppo di un server HTTP concorrente in linguaggio C, si è potuto mettere in pratica una vasta gamma di concetti teorici studiati durante il corso.

Oltre agli aspetti puramente tecnici, il progetto ha permesso di sviluppare soft skills fondamentali come la risoluzione dei problemi, il debugging, la gestione del tempo e l'autonomia nello studio e nella ricerca di soluzioni.

Infine, questo progetto, sebbene semplice nelle sue funzionalità, rappresenta un punto di partenza estremamente utile per l'evoluzione verso sistemi web più avanzati. Con l'integrazione delle funzionalità proposte negli sviluppi futuri, potrebbe diventare un server HTTP completo in grado di supportare servizi RESTful o microservizi, costituendo così un ottimo esempio di infrastruttura minima per applicazioni distribuite.

In conclusione, si ritiene che il progetto abbia soddisfatto appieno gli obiettivi didattici, fornendo una panoramica pratica e concreta delle tecnologie di rete, con margine per ulteriori esplorazioni in ambito di sicurezza, efficienza e scalabilità.